Difficulties in Parsing SGML

by
Jim Heath and Larry Welsch
National Bureau of Standards
Gaithersburg, Md. 20899

Abstract - A frequently cited problem with the Standard Generalized Markup Language (SGML) is that applications using the standard have been slow in arriving. Part of this delay is because of the instability of the standard and part because of constructs of the language that are functionally redundant and/or add unnecessary complexity to both machine and human processing. This paper is based on our experience implementing an SGML parser using commonly available tools for building programming language translators. It describes the problems we encountered and suggests modifications to SGML to eliminate those problems. The modified language can be implemented using well tested tools and will be more stable and more amenable to both computer and human processing while maintaining all of the fundamental strengths of SGML.

Introduction

The National Bureau of Standards (NBS), among other activities, promotes the acceptance of voluntary standards. Part of the acceptance requires that products be available to show users how the standard might be useful in their situation. If products are too slow in becoming available, users may have to accept solutions which do not use standards.

At NBS we are often asked to develop prototype applications to show the usefulness of a standard and, like other developers, are frustrated by some of the coding required. All too often, standards that are meant to be processed by computer based systems have been developed without regard to established conventions and models and without regard to the tools that might be used to build applications. The resultant standards are technically flawed, require substantial changes,[1] and are onerous to process. Additionally, constructs that appeared useful when the standard was being specified become obstacles to both computer and human processing.

The Office Systems Engineering Group at NBS has looked at SGML[2] and written a parser using common tools for language processing. In the process of doing this we have several suggestions for improvement to the language. Although the bulk of this paper focuses on parsing a part of an SGML document known as the document type declaration (DTD), the techniques and the conclusions are applicable to the entire SGML specification.

Some SGML Terminology and Notation

Although it is not within the scope of this paper to describe even a significant part of SGML, the following information will be helpful in understanding subsequent parts of this report. SGML is an International Standards Organization (ISO) standard and a subset of this standard has been proposed as a Federal Information Processing Standard (FIPS). A basic tenet of SGML is that markup[3] should be descriptive

---

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and / or specific permission.

© 1988 ACM 0-89791-291-8/88/0012/0071 $1.50

1. Exclusive of glossary, overview, and appendices, the SGML standard, ISO 8879-1986(E), is approximately 38 pages in length. Since its release in October of 1986, there have been 13 pages of technical amendments (ISO 8879-1986(E) Amendment 1) which have been added to "improve the expression of the design of SGML."
2. *ISO 8879-1986 Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML).* (Geneva, Switzerland: International Standards Organization), (1986).
3. Markup is defined in ISO 8879-1986(E) as "Text that is added to the data of a document to convey information about it."

rather than procedural. Descriptive markup is characterized by nouns (e.g., ´paragraph´, ´chapter´, etc.) while procedural markup is characterized by verbs (e.g., ´skip 2 lines´, ´go to next page´, etc.). An SGML document consists of several parts including what is commonly called the document type declaration (DTD) and the document instance. The DTD provides a template that describes the content and structure of the document instance. Among other things, the DTD can contain entity declarations and entity references. Entity declarations are similar to macro definitions in other languages and entity references are similar to macro substitutions. As a simple example, if ´x´ is defined as:

```
<!ENTITY % x "Some Text">
```

then whenever ´x´ is referenced as an entity as in

```
%x;
```

it is replaced with

```
Some Text
```

Another declaration that is found in the DTD is the element declaration. A simple element declaration would be:

```
<!ELEMENT memo (date,subj,body) >
```

The element declaration contains a production called a content model (the part in parentheses in the above example) which describes the content for the element being declared. If there is more than one content token in the content model, then each content token is separated by a ´connector´ which is either ´,´, ´|´, or ´&´:

. - is the sequence operator and (A , B) means that A then B must occur.

| - is the **or** operator and (A | B) means that either A or B (but not both) must occur.

& - is the **and** operator and (A & B) means that both A and B must in any order.

Each content token may have an optional occurrence indicator which is one of ´?´, ´+´, or ´*´ where these have the same meanings normally found in regular expressions, i.e.:

? - means 0 or 1 times.

+ - means 1 or more times.

* - means 0 or more times.

General Approach

As described earlier, one component of an SGML document is the document type declaration (DTD) - this component provides a template that describes the structure and content of the document instance. The DTD is described by a set of productions that appear consistent with those of a context-free grammar; a closer look suggests that much of the DTD may be described by a regular grammar. To test this idea, we decided to use **lex**[4] to build a recognizer, consisting of a lexical analyzer and a parser[5], for an SGML DTD. This forced us to formally specify the grammar and clearly identified those parts that could not be described (or not described completely) by a regular grammar. By analyzing the the limitations and capabilities of our recognizer[6] we were able to determine which parts of SGML were awkward from the developer's viewpoint; not surprisingly, these same constructs are awkward for human processing of SGML documents. We tried to absolutely minimize the amount of code that we developed for **lex**. We put as little as possible into the program fragments that are added to **lex** programs and instead tried to exploit the capability of **lex** to recognize regular expressions. We did use *start* conditions as supported by **lex** (largely to handle overlapping token classes[7]) and we were able to simplify processing by developing a library of lex generated functions. These two ideas are discussed in the appendix of the paper and should be read by anyone trying to follow our method.

Some Suggested Improvements for SGML

As a result of building the parser (and working with some real applications of SGML), we suggest several changes. The following are a few of the changes that will improve SGML.

Remove Artificial Restrictions on Entity References

SGML imposes some arbitrary restrictions on entity references. Instead of requiring that a reference be recognizable and the data stream

---

4. M. E. Lesk and E. Schmidt, *Lex - A Lexical Analyzer Generator*. (Murray Hill, New Jersey: Bell Laboratories), (1975).

5. In language processing, the typical use of **lex** is as a program generator for a lexical scanner, the generated program does, however, recognize regular expressions and hence can be used to parse sentences of a regular language; in our recognizer for the DTD, we used **lex** for both the lexical analyzer and parser functions!

6. Although our recognizer was written with **lex** and did not use a generator for a push-down automaton, the parts thats that we identified as awkward would not have been helped by using a context-free grammar as a model. It is important to remember that the recognizer that we are describing is for the DTD only - not for the document instance.

7. We will return to overlapping token classes later in the paper.

resulting from its expansion be syntactically correct, SGML places additional restrictions depending on the definition of the entity. These restrictions make construction of an entity preprocessor impractical and, during processing of an SGML document, require a high level of coupling between the scanner and the parser. For example, a parser cannot determine (based on the input token stream) whether the following describes a legal content model or not.

```
grpo name Ee grpc
```

If the above token stream came from the following data stream

```
<!ENTITY % test "abc">
...
...
<!ELEMENT test (%abc)>
```

then it would be valid but if it resulted from the following data stream

```
<!ENTITY % test "(abc">
...
...
<!ELEMENT test %abc)>
```

then it would be in error. In contrast, the C programming language expands macros wherever they are recognized; allowing for macro expansion to be performed by a preprocessor.[8] **SGML should borrow from the C programming language - the only requirement placed on entity references is that they be recognizable[9].**

Remove Start and End Tag Minimization

The construct of start and end tag minimization places a high burden on both humans and programs that process SGML documents. Although the idea of tag minimization may seem reasonable (at least for human processing) for simple documents, it becomes unwieldy and confusing in documents encountered in the real world and the few keystrokes that are saved are more than offset by the confusion that comes from looking at a document with omitted tags. The combination of intelligent editors (which fully tag a document as it is created) along with other forms of minimization (e.g., short references) obviates the need for tag minimization.

As an example of the awkwardness of this construct on a parser, the following two content models (which reduce to equivalent regular expressions) require different processing because of tag minimization.

```
((A , B) | B)
```

```
(A? , B)
```

Start and end tag minimization should be removed from SGML.

Eliminate Overlapping Token Classes

The common approach to language processing is for the lexical scanner to pass tokens to the parser without regard for context. This is difficult in SGML because of overlapping token classes, e.g., "999" can be a "number," a "number token", or a "name token." Many of these token classes of SGML seem completely arbitrary. **In the absence of convincing arguments to the contrary, the number of token classes in SGML should be reduced to nonoverlapping token classes.**

Separators

Separators are a problem in SGML. As noted by Kaelbling[10], the separators that are supposed to be ignored are written into the very productions that should ignore them. To further confuse the issue, SGML defines multiple (and overlapping) classes of separators. No arguments to support this are given and human and machine processability would be improved by having only a single class. It is also puzzling that carriage returns (CR) and line feeds (LF) are not treated consistently in SGML - in some productions, either is acceptable, in others only CR is allowed (a real nuisance for systems that tend to end lines only with LF). **One class of separators should be defined for SGML. CR and LF (in the absence of a cogent argument otherwise) should be interchangeable in any token class in which they occur.**

Remove an Unneeded Connector

There exists in SGML a connector **&** which is defined as follows:

A & B ::= (A , B) | (B , A)

While this appears reasonable and does provide a model for situations where parts of a document are meant to be kept together without regard to order (such as figure and associated caption), this same situation can easily be handled by using just the "," and "|" connectors. Admittedly, this technique fails when large numbers of elements are involved since the number of productions

---

8. For those who would argue that the current requirement that an entity reference resolve to a complete parameter eliminates obfuscatory references, consider the following example which is legal according to the current standard.

```
<!ENTITY % lpart "(abc">
<!ENTITY % rpart ")">
<!ENTITY % abc "%lpart;%rpart">
...
...
<!ELEMENT test %abc>
```

10. On Improving SGML, Michael J. Kaelbling, OSU-CISRC-7/88-TR22

associated with the following model

$$A_1 \& A_2 \& \ldots \& A_n$$

is n *factorial* (n!) but this doesn't seem to be a problem since we have never seen a real instance of an **AND** group with more than three elements[11].

Conclusions

We have identified some areas where SGML could be improved for both human and machine processing without sacrificing any of its fundamental strengths. Others studies of SGML have resulted in similar findings. In particular, projects[12] at The Ohio State University suggest several improvements to SGML that are consistent with what we have presented. Taken together with the experiences of real users of SGML, the standard could be improved to the benefit of everyone.

SGML is a language intended for processing with computer based systems, to this end, it should follow an established grammar model and be processable with existing tools. Functionally redundant or unused constructs should be removed from the language. The conclusions of this paper are applicable not only to SGML but to all language standards. We encourage the standards community to consider applying techniques such as those described in this paper. The result will be more stable and elegant standards supported by a wealth of well written applications.

Appendix

Introduction

The appendix explains how we constructed the SGML DTD parser. The next section provides some definitions for the terms that we use. The parser was constructed from a heirarchy of coupled finite state acceptors. Section 3 describes how we coupled the finite state acceptors together, and the implications of coupling.

Definitions

Although space does not permit a complete discussion, the following definitions are given so that readers will have a common understanding of the terms we use. Some of the concepts are difficult to rigorously define in simpler terms and the definitions should be regarded as providing only an intuitive notion of their meaning.

- alphabet - a finite nonempty set of symbols.
- string - a (possibly empty) sequence of symbols from the alphabet.
- grammar - a finite nonempty set of rules (or productions) which specifies the syntax of a language.
- regular expression - for the purposes of this paper, regular expressions will be considered to be those expressions which are acceptable as patterns to **lex**.
- regular grammar -
  - — A set of symbols known as *terminal* symbols.
  - — A set of symbols known as *nonterminal* symbols.
  - — A set of productions where each production consists of a nonterminal symbol called the *left side* of the production, an arrow, and either a terminal or a terminal followed by a nonterminal symbol called the *right side* of the production.
- context free grammar[13] -
  - — A set of symbols known as *terminal* symbols.
  - — A set of symbols known as *nonterminal* symbols.
  - — A set of productions where each production consists of a nonterminal symbol called the *left side* of the production, an arrow, and a sequence of terminal and/or nonterminal symbols called the *right side* of the production.
  - — A designation of one of the nonterminal symbols as the *start* symbol.

---

11. The DTD for MIL-STD 28001 (in the version we reviewed) used very few **AND** groups and never with more than two elements in a group; the sample DTD's published by the Association of American Publishers does not seem to use **AND** groups at all.

12. Michael J. Kaelbling, *On Improving SGML.* (The Ohio State University, Columbus Ohio), (1988).
Michael J. Kaelbling, *Braced Languages and a Model of Translation for Context-Free Strings: Theory and Practice* (The Ohio State University, Columbus Ohio), (1988).

13. Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools.* (Reading, Massachusettes: Addison Wesley), (1986).

- context sensitive grammar -
  - — A set of symbols known as *terminal* symbols.
  - — A set of symbols known as *nonterminal* symbols.
  - — A set of productions where each production consists of a sequence of terminal and/or nonterminal symbols called the *left side* of the production, an arrow, and a sequence of terminal and/or nonterminal symbols called the *right side* of the production.
  - — The number of symbols in the *left side* of the production is always less than or equal to the number of symbols on the *right side* of the production.
- language - a subset of the possible sequences of symbols (string) of an alphabet. In the case of a finite language it is possible to specify by exhaustive enumeration all of the strings which are its elements. More commonly, however, this complete enumeration is not possible and the language is specified by a finite set of rules or *productions* called a grammar.

Coupling Finite State Acceptors Together

This section describes the coupling of FSAs together to form larger FSAs; although this concept is described in texts, we have not often seen it in practice. We describe a method for coupling FSA's in a **lex** generated program, the meaning of the coupling, then the advantages derived from such a coupling, and finally why we chose to couple FSAs directly together as opposed to using pushdown automaton as an interface. Compilers typically use a pushdown automaton.[14]

Coupling **lex** *generated FSAs*

One of the central ideas of our recognizer is that one **lex** generated program can call another **lex** generated program as a function. While **lex** does allow the user to define substitution strings, we often preferred to call another FSA for two reasons:

— when substitutions were used extensively, the resulting programs became large, slow to generate, and slow to compile.

— by calling a function as an action instead of just placing the the substituted string in the pattern to be recognized, more information could be easily made available to the scanner.

For example, the parameter separator (ps) of SGML (similar to white space in the C programming language) appears often in SGML productions and is described by a moderately complex regular expression. Suppose **getps** is a **lex** generated program that returns 0 or 1 depending on whether or not it found some number of consecutive parameter separators (a useful operation in parsing DTD's) and the function runFSA() invokes a **lex** generated program and returns the result to the calling function. Then the sample code fragment[15] given below could be used to recognize the first part of an SGML element declaration.

```
"<!ELEMENT"{
            if (runFSA("getps") <= 0)
              printf("ERROR - expected ps+\n");
            else
              BEGIN GET_ELEMENT_TYPE;
            }
```

The Meaning of Coupling Finite State Acceptors

To consider the meaning let us take an example from SGML. Assume that there is an FSA that can recognize and accept element declarations, called Π. Assume further that there is an FSA that can recognize and accept entity declarations, called π. Let Γ be the start state and Ω be the end state for the element declaration FSA, Π. Further let γ be the start state and ω be the end state for the entity declaration FSA,π, both shown in figure 1.

---

14. Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilers, Principles, Techniques, and Tools*. (Reading, Massachusettes: Addison Wesley), (1986).

    F. R. A. Hopgood, *Compiling Techniques*. (New York, N.Y.: American Elsevier Publishing Company, Inc.), (1969).

    Alfred V. Aho and Jeffrey D. Ullman, *The Theory of Parsing, Translation and Compiling*. vol 1,2: (Englewood Cliffs, N.J: Prentice Hall, Inc.), (1972).

15. The code samples given in this paper were written for the purpose of explanation - they are not optimal and may not allow for all input. The actual code of our parser uses similar techniques but is somewhat different.

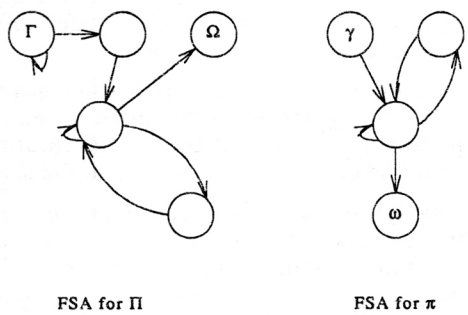

Figure 1. These are entity and element declaration FSAs.

Assume there exists an FSA Δ that can decide whether or not a particular SGML construct must be an entity declaration or an element declaration before scanning the construct. These three FSAs may be coupled as shown in figure 2.

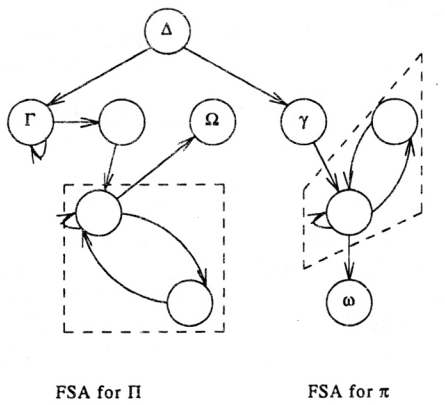

Figure 2. This shows how decider FSA can choose between two FSAs.

The coupling is accomplished in the state in Δ that decides that if the pattern is correct then the pattern must be either an entity declaration or an element declaration and that arc is then connected to the appropriate start state of either Π or π. The end states go to the appropriate next state in Δ.

Note that the two-state combination shown in the figure 3 are repeated in both the entity declaration and element declaration state machine shown in figure 2. This state combination could be the part of both FSAs that recognizes identifiers. Rather than repeat this we might consider the two state identifier FSA to be a separate identifier FSA ι.

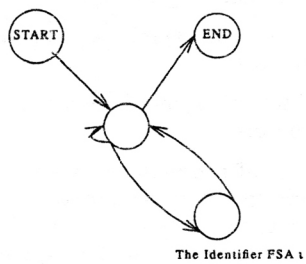

Figure 3. This shows the common state combination in the boxes shown in the previous figure.

Unlike the entity and element FSAs the identifier FSA when coupled does not always return to the same state after it completes. Rather, the identifier FSA must return to a different state: the state that it must normally goto if there is a success. This coupling may be modeled as shown in figure 4.

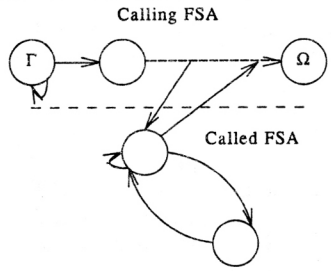

Figure 4. This shows how to link two FSAs together.

Advantages of Coupling

There are two advantages to coupling FSAs as described above. The first is that each FSA corresponds to an abstraction that is "close to the problem to be solved."[16] For example, entity declarations, element declarations and notation declarations are each separate FSAs that correspond to a particular description in the SGML standard.

The second advantage is that the total number of states that need to be represented will be reduced. Working with **lex** we discovered that when more than on the order of 15 programmer

16. Bjarne Stroustrup, *The C++ Programming Language*. p. 6, (Reading, Massachusettes: Addison-Wesley Publishing Company), (1986).

defined states were created that **lex** tended to fail. By reducing the total number of states that need to be maintained by any component of the parser we were able to stay within the bounds of the tools that we had readily available and use the tool we felt lent itself to building the parser.[17]

The resulting cost benefits of having a binding of FSA to SGML component over the life of the standard will be significant. For example Weinberg estimates that 50% of the cost of a program is spent on maintenance and enhancements, 17% on developmental analysis and design, and 25% on developmental testing, and 8% in coding.[18] The abstraction described here will reduce the costs in all areas of the program's life cycle.

Why not a Pushdown Automaton

Two issue that needs to be addressed are

- why we did not choose a pushdown automaton to build the parser and
- by introducing procedure calls of FSAs within FSAs have we not introduced a pushdown automaton.

The answer to the second question is no - in our case, the FSAs do not ever call each other in an iterative fashion.

The answer to the first question is historical. We were first interested in determining how much of an SGML DTD parser could be implemented using **yacc** and **lex** and what parts would require more. What we discovered was that there was very little that could not be done using **lex** alone; further, **yacc** brought only a slight gain[19]. By looking at those parts of the parser that could not be implemented without using memory in addition to the FSAs implemented in **lex**, we were able to determine what parts of an SGML DTD parser could not easily be implemented as an FSA and therefore cause the grammar of SGML DTDs to be a context-free, context-sensitive or unrestricted[20] grammar.[21]

We discovered that those parts of the SGML DTD that we could not parse with an FSA were context-sensitive[22]. Hence, there was no gain by using a push down automaton and, therefore, **yacc** to implement the parser.

---

17. In layman's terms we do not use screw drivers to pry open windows.
18. Victor Weinberg, *Structured Analysis*. p. 30, (Englewood Cliffs, N.J: Prentice Hall, Inc.), (1980).
19. The content model of an element declarations allows nesting and therefore could not be processed by **lex** alone but could be processed by **yacc**.
20. Sometimes referred to as type 0 or phrase structure.
21. John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. chap. 9, (Reading, Massachusettes: Addison-Wesley Publishing Company), (1979).
22. With the one exception noted above.